Sacado: Automatic Differentiation Tools for C++ Codes

Eric Phipps and David Gay

Sandia National Laboratories

Trilinos User’s Group Meeting
November 6, 2007

SAND 2007-7122 P
What is Automatic Differentiation (AD)?

• Technique to compute analytic derivatives without hand-coding the derivative computation

• How does it work -- freshman calculus
  – Computations are composition of simple operations (+, *, sin(), etc…) with known derivatives
  – Derivatives computed line-by-line, combined via chain rule

• Derivatives accurate as original computation
  – No finite-difference truncation errors

• Provides analytic derivatives without the time and effort of hand-coding them

\[ y = \sin(e^x + x \log x), \ x = 2 \]

<table>
<thead>
<tr>
<th>( x )</th>
<th>( \frac{d}{dx} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.000</td>
<td>1.000</td>
</tr>
<tr>
<td>7.389</td>
<td>7.389</td>
</tr>
<tr>
<td>0.301</td>
<td>0.500</td>
</tr>
<tr>
<td>0.602</td>
<td>1.301</td>
</tr>
<tr>
<td>7.991</td>
<td>8.690</td>
</tr>
<tr>
<td>0.991</td>
<td>-1.188</td>
</tr>
</tbody>
</table>
Sacado: AD Tools for C++ Codes

- Sacado provides several modes of Automatic Differentiation (AD)
  - Forward (Jacobians, Jacobian-vector products, …)
  - Reverse (Gradients, Jacobian-transpose-vector products, …)
  - Taylor (High-order univariate Taylor series)

- Sacado implements AD via operator overloading and C++ templating
  - Expression templates for OO efficiency
  - Application code templating for easy incorporation

- Designed for use in large-scale C++ codes
  - Apply AD at “element-level”
  - Very successful in Charon application code
  - Sacado::FEApp example demonstrates approach

- Sacado provides other useful utilities
  - Scalar flop counting (Ross Bartlett)
  - Scalar parameter library
  - Template utilities
The Usual Suspects

- Configure options
  --enable-sacado — Enables Sacado at Trilinos top-level
  --enable-sacado-tests, --enable-tests — Enables unit, regression, and performance tests
    --with-cppunit-prefix=[path] — Path to CppUnit for unit tests
    --with-adolc=[path] — Enables Taylor polynomial unit tests with ADOL-C
  --enable-sacado-examples, --enable-examples — Enables examples
    nox/examples/epetra/LOCA_Sacado_FEApp — Continuation example using Sacado::FEApp 1D finite element application

- Mailing lists
  Sacado-announce@software.sandia.gov, Sacado-checkins@software.sandia.gov,
  Sacado-developers@software.sandia.gov, Sacado-regression@software.sandia.gov,
  Sacado-users@software.sandia.gov

- Bugzilla: http://software.sandia.gov/bugzilla
- Bonsai: http://software.sandia.gov/bonsai/cvsqueryform.cgi
- Web: http://software.sandia.gov/Trilinos/packages/sacado (not much there yet)
- Doxygen documentation (not all that useful)
- Examples are best way to learn how to use Sacado
#include "Sacado.hpp"

// The function to differentiate
template<typename ScalarT>
ScalarT func(const ScalarT& a, const ScalarT& b, const ScalarT& c) {
  ScalarT r = c*std::log(b+1.)/std::sin(a);
  return r;
}

int main(int argc, char **argv) {
  double a = std::atan(1.0);                           // pi/4
  double b = 2.0;
  double c = 3.0;
  int num_deriv = 2;                                    // Number of independent variables

  // Fad objects
  Sacado::Fad::DFad<double> afad(num_deriv, 0, a);   // First (0) indep. var
  Sacado::Fad::DFad<double> bfad(num_deriv, 1, b);   // Second (1) indep. var
  Sacado::Fad::DFad<double> cfad(c);                  // Passive variable
  Sacado::Fad::DFad<double> rfad;                     // Result

  // Compute function
  double r = func(a, b, c);

  // Compute function and derivative with AD
  rfad = func(afad, bfad, cfad);

  // Extract value and derivatives
  double r_ad = rfad.val();   // r
  double drda_ad = rfad.dx(0); // dr/da
  double drdb_ad = rfad.dx(1); // dr/db
Differentiating Element-Based Codes

- Global residual computation (ignoring boundary computations):

\[ f(x) = \sum_{i=1}^{N} Q_i^T e_{k_i}(P_ix) \]

- Jacobian computation:

\[ \frac{\partial f}{\partial x} = \sum_{i=1}^{N} Q_i^T J_{k_i} P_i, \quad J_{k_i} = \frac{\partial e_{k_i}}{\partial x_i}, \quad x_i = P_ix \]

- Jacobian-transpose product computation:

\[ w^T \frac{\partial f}{\partial x} = \sum_{i=1}^{N} (Q_iw)^T J_{k_i} P_i \]

- Hybrid symbolic/AD procedure
  - Element-level derivatives computed via AD
  - Exactly the same as how you would do this “manually”
  - Avoids parallelization issues
Impacts of AD in Charon

- Mobile Defects
- Oxide Defects
- Dynamical Defects
- Oxide Physics
- Multi-Trap SRH
- SRH
- Drift-Diffusion

**PHYSICS**

**Jacobian Computation**

- Finite Difference
- Forward AD

**Jacobian-Transpose Product Computation**

- Reverse AD

**Cross section (Parameter 3)**

\[ e^- + V^0 \rightarrow V^- \]

**Cross section (Parameter 23)**

\[ h^+ + PV^- \rightarrow PV^0 \]

**Experimental Data**

- Initial Calibration
- Optimized Fit

**Voltage**

- Current

- DOF Per Element (4*N)

- Relative Eval. Time

- Scaled Sensitivity

- Parameter
Where Sacado is going in the future

• Documentation
  – Website, tutorials, papers, etc…

• Performance improvements
  – Expression level reverse-mode (Sacado::ELRFad)

• Leveraging AD technology for intrusive uncertainty quantification
  – Polynomial chaos expansions via operator overloading

• Impacting more applications
  – Using Sacado is more about software engineering than AD

• SESS presentation 11/13/07
  – More in-depth tutorial on using Sacado in applications